A Domain-Specific Language for Programming in the Tile Assembly Model*

David Doty† Matthew J. Patitz‡

Abstract

We introduce a domain-specific language (DSL) for creating sets of tile types for simulations of the abstract Tile Assembly Model. The language defines objects known as tile templates, which represent related groups of tiles, and a small number of basic operations on tile templates that help to eliminate the error-prone drudgery of enumerating such tile types manually or with low-level constructs of general-purpose programming languages. The language is implemented as a class library in Python (a so-called *internal DSL*), but is presented independently of Python or object-oriented programming, with emphasis on supporting the creation of visual editing tools for programmatically creating large sets of complex tile types without needing to write a program.

1 Introduction

Erik Winfree [17] introduced the abstract Tile Assembly Model (aTAM) as a simplified mathematical model of molecular self-assembly. In particular, it attempts to model Seeman's efforts to coax DNA double-crossover molecules to self-assemble, programmable through the careful selection of sticky ends protruding from the sides of the double-crossover molecules. The basic component of the aTAM is the *tile type*, which defines (many identical copies of) a square tile that can be translated but not rotated, which has glues on each side of strength 0, 1, or 2 (when the temperature is set to 2), each with labels, so that abutting tiles will bind with the given strength if their glue labels match. In particular, by setting the temperature to 2, we may enforce that the abutting sides of two tiles with strength-1 glues provide two input "signals" which determine the tile types that can attach in a given location. Such enforced cooperation is key to all sophisticated constructions in the aTAM.

1.1 Background

The current practice of the design of tile types for the abstract tile assembly model and related models is not unlike early machine-level programming. Numerous theoretical papers [6,13–15] focus on the *tile complexity* of systems, the minimum number of tile types required to assemble certain structures such as squares. A major motivation of such complexity measures is the immense level of laboratory effort required (presently) to create a physical implementation of a tile.

*This research was partially supported by NSF grants 0652569 and 0728806.

†Department of Computer Science, Iowa State University, Ames, IA 50011, USA. ddoty@iastate.edu

‡Department of Computer Science, Iowa State University, Ames, IA 50011, USA. mpatitz@cs.iastate.edu.

Early electronic computers occupied entire rooms to obtain a fraction of the memory and processing power of today's cheapest mobile telephones. This limitation did not stop algorithm developers from creating algorithms, such as divide-and-conquer sorting and the simplex method, that find their most useful niche when executed on data sets that would not have fit on all the memory existing in the world in 1950. Similarly, we hope and expect that the basic components of self-assembly will one day, through the ingenuity of physical scientists, become cheap and easy to produce, not only in quantity but in variety. The *computational* scientists will then be charged with developing disciplined methods of organizing such components so that systems of high complexity can be designed without overwhelming the engineers producing the design. In this paper, we introduce a preliminary attempt at such a disciplined method of controlling the complexity of designing tile assembly systems in the aTAM.

Simulated tile assembly systems of moderate complexity cannot be produced by hand; those constructions that we have personally designed [4, 8, 10, 11] have all been produced as the output of a computer program that handles the drudgery of looping over related groups of individual tile types. However, even writing a program to produce tile types directly is error-prone, and more low-level than the ways that we tend to think about tile assembly systems.

Fowler [5] suggests that a *domain-specific language (DSL)* is an appropriate tool to introduce into a programming project when the syntax or expressive capabilities of a general-purpose programming language are awkward or inadequate for certain portions of the project. Fowler distinguishes between an *external DSL*, which is a completely new language designed especially for some task (such as SQL for querying and updating databases), and an *internal DSL*, which is a way of "hijacking" the syntax of an existing general-purpose language for expressing concepts specific to the domain (such as Ruby on Rails for writing web applications). An external DSL may be as simple as an XML configuration file, and an internal DSL may be as simple as a class library. In either case the major objective is to express "commands" in the DSL that more closely model the way one thinks about the domain than the "host" language in which the project is written.

If the syntax and semantics of the DSL are precisely defined, this facilitates the creation of a *semantic editor* (text-based or visual), in which programs in the DSL may be produced using a tool that can visually show semantically meaningful information such as compilation or even logical errors, and can help the programmer directly edit the abstract components of the DSL, instead of editing the source code directly. For example, Intellij IDEA and Eclipse are two programs that help Java programmers to do refactorings such as variable renaming or method inlining, which are at their core operations that act directly on the abstract syntax tree of the Java program rather than on the text constituting the source code of the program. We have kept such a tool in mind (although we have not yet implemented it) as a guide for how to appropriately structure the DSL for designing tile systems.

We structure this paper in such a way as to de-emphasize any dependence of the DSL on Python in particular or even on object-oriented class libraries in general. Eventually we will build a visual editor that mostly shields the tile set "programmer" from the Python roots of the implementation. The DSL provides a high-level way of *thinking* about tile assembly programming, which, like any high-level language or other advance in software engineering, not only automates mundane tasks better left to computers, but also *restricts* the programmer from certain error-prone tasks, in order to better guide the design, following the dictum of Antoine de Saint-Exupery that a design is perfected "not when there is nothing left to add, but nothing left to take away."

1.2 Brief Outline of the DSL

We now briefly outline the design of the tile assembly DSL. Section 2 provides more detail.

The most fundamental component of designing a tile assembly system manually is the tile type. In our DSL, the most fundamental component is an object known as a *tile template*. A tile template represents a group of tile types (each tile type being an *instance* of the tile template), sharing the same input sides, output sides, and function that transforms input signals into output signals. The two fundamental operations of the DSL are *join* and *add transition*. Both make the assumption that each tile template has well-defined input and output sides. In a join, an input tile template A is connected to an output tile template B in a certain direction $d \in \{N, S, E, W\}$, expressing that an instance t_A of A may have on its output side in direction d an instance t_B of B. This expresses an intention that in the growth of the assembly, t_A will be placed first, then t_B, and they will bind with positive strength, with t_A passing information to t_B. Whereas a join is an operation connecting the output side of a tile template to the input side of another, a transition "connects" input sides to output sides within a single tile template, by specifying how to compute information on the output side as a function of information on the input sides. This is called *adding* a transition rather than *setting*, since the information on the output sides may contain multiple independent output signals, and their computations may be specified independently of one another if convenient.

This notion of independent signals is modeled already in other DSLs for the aTAM [1] and for similar systems such as cellular automata [3]. The join operation, however, makes sense in the aTAM but not in a system such as a cellular automaton, where each cell contains the same transition function(s). The notion of tile templates allows one to break an "algorithm" for assembly into separate "stages", each stage corresponding to a different tile template, with each stage being modularized and decoupled from other stages, except through well-defined and restricted signals passed through a join. There is a rough analogy with lines of code in a program: a single line of code may execute more than once, each time executed with the state of memory being different than the previous. Similarly, many different tile types, with different actual signal values, generated from the same tile template, may be placed during the growth of an assembly. Another difference between our language and that of [1] is that our language appears to be more general; rather than being geared specifically toward the creation of geometric shapes, our language is more low-level, but also more general.[1]

This paper is organized as follows. Section 2 describes the DSL for tile assembly programming in more detail and gives examples of design and use. Section 3 concludes the paper and discusses future work and open theoretical questions. Due to space constraints, we refer the reader to [9],which contains a self-contained introduction to the Tile Assembly Model, for a formalism of the aTAM. More details and discussion may be found in [12,13,17]. A preliminary DSL implementation can be found at `http://www.cs.iastate.edu/~lnsa`.

2 Description of Language

The DSL is written as a class library in the Python programming language. It is designed as a set of classes which encapsulate the logical components needed to design a tile assembly system in the aTAM where the temperature value $\tau = 2$. The goal of the DSL is to abstract the tedious,

[1] An extraordinarily imprecise analogy would be that the progression *manual tile programming* → *Doty-Patitz* → *Becker* is roughly analogous to *machine code* → *C* → *Logo*.

low-level details of manually designing individual tile types, and thus free the developer to focus on the higher level design of larger modules.

The DSL is designed around the principle notion that data moves through an assembly as 'signals' which pass through connected tiles as the information encoded in the input glues, allowing a particular tile type to bind in a location, and then, based on the 'computation' performed by that tile type, as the resultant information encoded in the glues of its output edges. (Of course, tiles in the aTAM are static objects so the computation performed by a given tile type is a simple mapping of one set of input glues to one set of output glues which is hardcoded at the time the tile type is created.) Viewed in this way, signals can be seen to propagate as tiles attach and an assembly forms, and it is these signals which dictate the final shape of the assembly. Using this understanding of tile-based self-assembly, we designed the DSL from the standpoint of building tile assembly systems around the transmission and processing of such signals.

We present a detailed overview of the objects and operations provided by the DSL. We then demonstrate a full example exhibiting the way in which the DSL is used to design a tile set.

2.1 Client-side description

The DSL provides a collection of objects which represent the basic design units and a set of operations which can be performed on them. We describe these objects and operations in this section, without describing the underlying data structures and algorithms implementing them.

The DSL strictly enforces the notion of input and output sides for tile types, meaning that any given side can be designated as receiving a signal (an input side), sending a signal (an output side), or neither (a blank side).

2.1.1 DSL objects

Tile system The highest level object is the *tile system* object, which represents the full specification of a temperature 2 tile assembly system. It contains child objects representing the tile set and seed specification, and provides the functionality to write them out to files in a format readable by the ISU TAS simulator.

Tile In some cases, especially for tiles contained within seed assemblies, there is no computation being performed. In such situations, it may be easier for the programmer to fully specify the glues and properties of a tile type. The *tile* object can be used to easily create such hardcoded tile types.

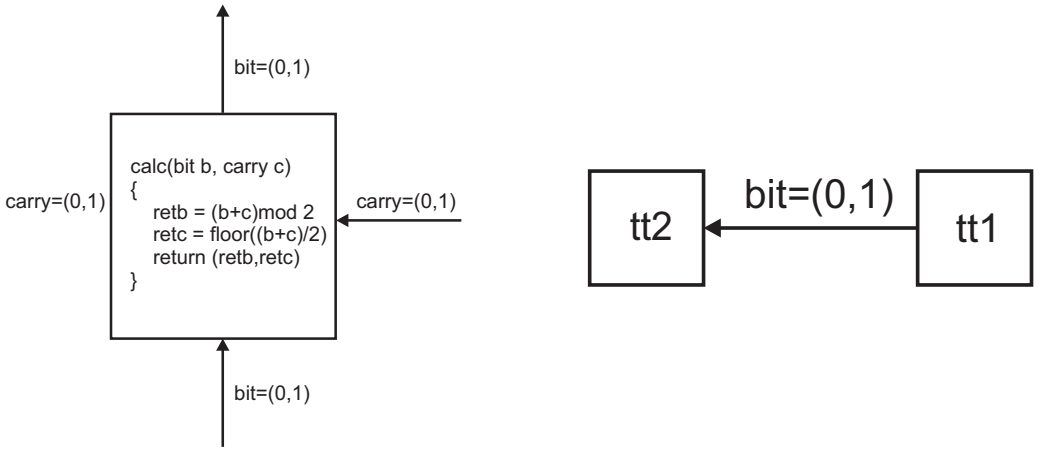
Tile template The principle design unit in the DSL is the tile template. A tile template is an object which represents a collection of tile types which share the following properties:

- They have exactly the same input, output, and blank sides.
- The types of signals received/transmitted on every corresponding input/output side are identical.
- The computation performed to transform the input signals to output signals can be performed by the same function, using the values specific to each instantiated tile type.

Logically, a tile template represents the set of tile types which perform the same computation on different values of the same input signal types.

Tile set template A *tile set template* contains all of the information necessary for generating a tile set. It contains the sets of tile and tile template objects which will be included in the tile set, as well as the logic for performing joins, doing error checking, etc. The tile set template object encapsulates information and operations that require communication between more than one tile template object, such as the *join* operation, whereas a tile template is responsible for operations that require information entirely local to the tile template, such as *add transition*.

Signal A *signal* is simply the name of a variable and the set of allowable values for it. For example, a signal used to represent a binary digit could be defined by giving it the name *bit* and the allowable values 0 and 1.



(a) Tansition: The function *calc* takes a bit and carry value as input from the bottom and right sides and computes new bit and carry values which are then output on the top and left sides.

(b) Join: The tile template on the right (tt1) is sending the signal *bit* with allowable values $(0, 1)$ to the tile template on the left (tt2).

Figure 1: Logical representations of some DSL objects

Transition *Transitions* are the objects which provide the computational abilities of tile templates. A transition is defined as a set of input signal names, output signal names, and a function which operates on the input signals to yield output signals. The logic of a function can be specified as a table enumerating all input signals and their corresponding outputs, a Python expression, or a Python function, yielding the full power of a general purpose programming language for performing the computations. An example is shown in Figure 1a.

2.1.2 DSL operations

Join The primary operation between tile templates, which defines the signals that are passed and the paths that they take through an assembly, is called a *join*. Joins are performed between the complementary sides (north and south, or east and west) of two tile templates (which need not be unequal), or a tile and a tile template. If one parameter is a tile, then all signals must be given just one value; otherwise a set of possible values that could be passed between the tile templates

is specified (which can still be just one). A join is directional, specifying the direction in which a signal (or set of signals) moves. This direction defines the input and output sides of the tile templates which a join connects. An example is shown in Figure 1b.

Add transition Transition objects can be added to tile template objects, and indicate how to compute the output signals of a tile template from the input signals. If convenient, a single transition can compute more than one output signal by returning a tuple. Each output signal of a tile template with more than one possible value must have a transition computing it.

Add chooser There may be multiple tile templates that share the same input signal values on all of their input sides. Depending on the join structure, the library may be able to use annotations (see below) to avoid "collisions" resulting from this, but if the tile templates share joins, then it may not be possible (or desirable) for the library to automatically calculate which tile template should be used to create a tile matching the inputs. In this case, a user-defined function called a *chooser* must be added so that the DSL can ensure that only a single tile type is generated for each combination of input values. This helps to avoid accidental nondeterminism.

Set property Additional properties such as the display string, tile color, and tile type concentrations can be set using user-defined functions for each property.

2.2 Additional features

The DSL provides a number of additional, useful features to programmers, a few of which will be described in this section. First, the DSL performs an analysis of the connection structure formed between tile templates by joins and creates *annotations*, or additional information in the form of strings, which are appended to glue labels. These annotations ensure that only tile types created from tile templates which are connected by joins can bind with each other in the directions specified by those joins, preventing the common error of accidentally allowing two tiles to bind that should not because they happen to communicate the same information in the same direction as two unrelated tiles.

Although some forms of 'accidental' nondeterminism are prevented by the DSL, it does provide methods by which a programmer can intentionally create nondeterminism. Specifically, a programmer can either design a chooser function which returns more than one output tile type for a given set of inputs, or one can add *auxiliary inputs* to a tile template, which are input signals that do not come from any direction.

The DSL provides additional error-checking functionality. For example, each tile template must have either one strength-2 input or two strength-1 inputs. As another example, the programmer is forced to specify a chooser function if the DSL cannot automatically determine a unique output tile template, which is a common source of accidental nondeterminism in tile set design.

2.3 Example construction

In this section, we present an example of how to use the DSL to produce a tile assembly system which assembles a log-width binary counter. In order to slightly simplify this example, the seed row of the assembly, representing the value 1, will be two tiles wide instead of one. All other rows will be of the correct width, i.e. a row representing the value n will be $\lceil \log_2(n+1) \rceil$ tiles wide.

Figure 2 shows a schematic diagram representing a set of DSL objects, namely tiles, tile templates and joins, which can generate the necessary tile types. The squares labeled *lsbseed* and *msbseed* represent hard-coded tiles for the seed row. The other squares represent tile templates, with the names shown in the middle. The connecting lines represent joins, which each have a direction specified by a terminal arrow and a signal which is named (either *bit* or *carry*) and has a range of allowable values (either 0, or 1, or both). The dashed line between *lsbseed* and *msbseed* represents an implicit join between these two hard-coded tile types because they were manually assigned the same glues.

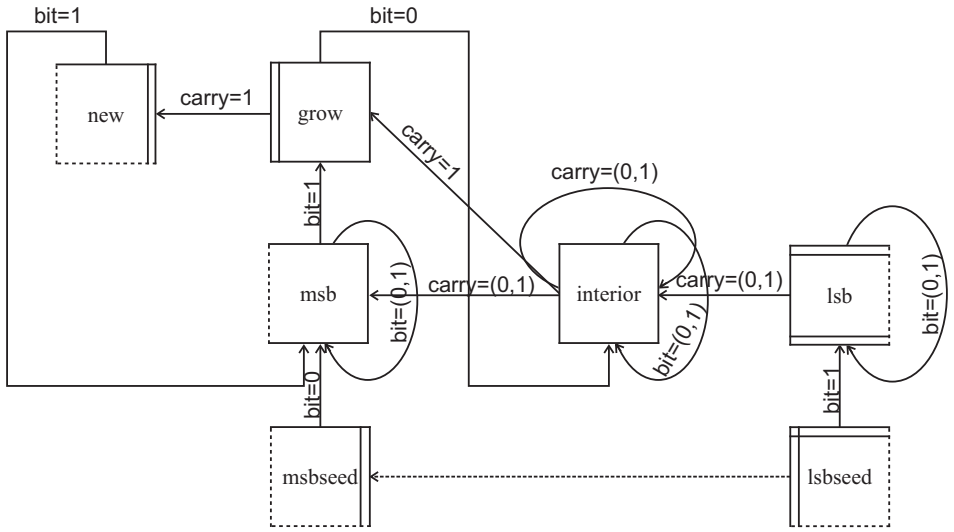


Figure 2: Schematic diagram depicting the tile templates and joins which are used to generate a tile set that self-assembles a log-width binary counter. All east-west joins pass the *carry* signal and south-north joins pass the *bit* signal (although some of them are restricted to subsets of the allowable values $(0, 1)$).

In addition to the joins, two more objects must be added to the tile templates to complete the necessary specifications. Transition functions must be added to the *lsb*, *iterior*, and *msb* tile templates to determine how the multiple input signal values are mapped to output signal values. Finally, since the tile templates *msb* and *grow* can both receive the signal bit=1 from *msb* and carry=1 from *interior*, a chooser function must be added to one of them so that the DSL can ensure that only a single type is generated for that situation.

The following code gives the full implementation of this example in the DSL. The first four arguments to each join method are respectively the strength, direction, "from" tile template, and "to" tile template, and the remaining arguments use Python keyword arguments to specify signals. This examples passes only one signal name through any side, but in general, multiple independent signal names can be specified.

```
lsbseed = tam.Tile(name='lsbseed', label='1', tilecolor='red', textcolor='black', westglue=('seed', 2))
msbseed = tam.Tile(name='msbseed', label='0', tilecolor='red', textcolor='black', eastglue=('seed', 2))

lsb = tam.TileTemplate(name='lsb')
interior = tam.TileTemplate(name='int')
msb = tam.TileTemplate(name='msb')
grow = tam.TileTemplate(name='grow')
```

```
new = tam.TileTemplate(name='new')

tst = tam.TileSetTemplate()

tst.join(2, N, lsbseed, lsb, bit=1)
tst.join(1, N, msbseed, msb, bit=0)
tst.join(2, N, lsb, lsb, bit=(0,1))
tst.join(1, W, lsb, interior, carry=(0,1))
tst.join(1, N, interior, interior, bit=(0,1))
tst.join(1, W, interior, interior, carry=(0,1))
tst.join(1, N, grow, interior, bit=0)
tst.join(1, W, interior, msb, carry=(0,1))
tst.join(1, W, interior, grow, carry=1)
tst.join(1, N, msb, msb, bit=(0,1))
tst.join(1, N, msb, grow, bit=1)
tst.join(2, W, grow, new, carry=1)
tst.join(1, N, new, msb, bit=1)

def nextBitAndCarry(bit, carry):
    return ((bit + carry) % 2 , (bit + carry) // 2)

# three ways to specify transition function
interior.addTransition(inputs=('bit', 'carry'), outputs=('bit', 'carry'), function=nextBitAndCarry)
lsb.addTransition(inputs=('bit'), outputs=('bit', 'carry'), table={0:(1,1), 1:(0,1))
msb.addTransition(inputs=('bit','carry'), outputs=('bit'), expression='(bit+carry)%2')

tst.setChooser(grow, inputs=('bit','carry'), expression='"grow" if bit == 1 and carry == 1 else "msb"')

lsb.setLabelFunction(outputs=['bit'], inputs=[], expression='str(bit)')
interior.setLabelFunction(outputs=['bit'], inputs=[], expression='str(bit)')
msb.setLabelFunction(outputs=['bit'], inputs=[], expression='str(bit)')
grow.setLabelFunction(outputs=['bit'], inputs=[], expression='str(bit)')
new.setLabelFunction(outputs=['bit'], inputs=[], expression='str(bit)')

tiles = tst.createTiles()
```

3 Conclusion and Future Work

We have described a domain-specific language (DSL) for creating tile sets in the abstract tile assembly model. This language is currently implemented as a Python class library but is framed as a DSL to emphasize its role as a high-level, disciplined way of thinking about the creation of tile systems.

3.1 Semantic Visual Editor

The DSL is implemented as a Python class library, but one thinks of the "real" programming as the creation of visual tile templates and the joins between them, as well as the addition of signal transitions. We intend to implement a visual editor that removes the Python programming and allows the direct creation of tile templates, and the execution of the other operations, from within the editor. It will detect and report errors, such as a tile template having only one input side of strength 1, while temporarily allowing the editing to continue. Other types of errors, that do not aid the user in being allowed to persist, such as the usage of a single side as both an input and output, are prohibited outright.

One can therefore create a tile set by directly drawing the tile templates as shown in Figure 2, while receiving helpful tips from a semantically-aware editor about errors.

3.2 Avoidance of Accidental Nondeterminism

Initially, our hope had been to design a DSL with the property that it could be used in a straightforward way to design a wide range of existing tile assembly systems, but was sufficiently restricted that it could be guaranteed to produce a deterministic tile assembly system, so long as nondeterminism was not directly and intentionally introduced by the programmer through chooser functions or auxiliary inputs. Alas, this is not the case.

In fact, one cannot even hope for the more modest goal of stating that the DSL sufficiently restricts tile assembly systems so that the undecidable problem of detecting whether a given tile set is deterministic becomes decidable through static analysis of the join structure of the tile templates. It it straightforward to use the DSL to design a TAS that begins the parallel simulation of two copies of a Turing machine so that if the Turing machine halts, two "lines" of double-bonded tiles are sent growing towards each other. The tiles comprising these two lines then meet and compete nondeterministically if and only if the TM halts.

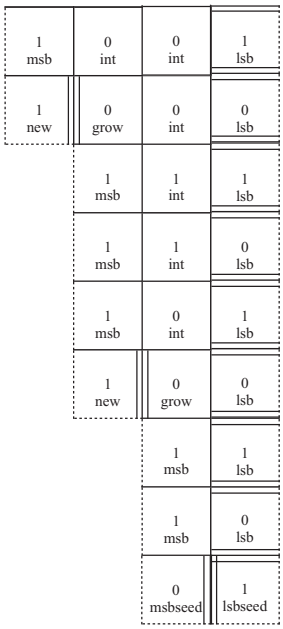


Figure 3: The first 9 rows of the assembly of the log-width binary counter. Note that each tile is labeled with its bit value on top and the name of the tile template from which it was generated on the bottom.

A common cause of "accidental nondeterminism" in the design of tile assembly systems is the use of a single side of a tile type as both an input and an output; this sort of error is prevented by the nature of the DSL in assigning each tile template unique, unchanging sets of input sides and output sides. But we cannot see an elegant way to restrict the DSL further to automatically prevent or statically detect the presence of the sort of "geometric nondeterminism" described in the Turing machine example. The development of such a technique would prove a boon to the designers of tile assembly systems. Conversely, perhaps there is a theorem analogous to that of Blum [2], which implies that any programming language in which all programs are guaranteed to halt requires uncomputably large programs to compute some functions which are trivially computable in a general-purpose language such as Python. If this is true, then accidental nondeterminism in tile assembly, like accidental infinite loops in software engineering, would be an unavoidable fact of life, and we would do better to spend time developing formal methods for proving determinism rather than hope that it could be automatically guaranteed.

3.3 Other Self-Assembly Models

The abstract Tile Assembly Model is a simple but powerful tool for exploring the theoretical capabilities and limitations of molecular self-assembly. However, it is an (intentionally) overly-simplified model. Generalizations of the aTAM, such as the kinetic Tile Assembly Model [16, 18], and alternative models, such as graph-based self-assembly [7], have been studied theoretically and implemented practically. We hope to leverage the lessons learned from designing the aTAM DSL to guide the design of more advanced DSLs for high-level programming in these alternative models.

Acknowledgement. We thank Scott Summers for help testing and using preliminary versions of the DSL.

References

[1] Florent Becker, *Pictures worth a thousand tiles, a geometrical programming language for self-assembly*, Theoretical Computer Science, to appear.

[2] Manuel Blum, *On the size of machines*, Information and Control **11** (1967), no. 3, 257–265.

[3] Hui-Hsien Chou, Wei Huang, and James A. Reggia, *The Trend cellular automata programming environment*, SIMULATION: Transactions of The Society for Modeling and Simulation International **78** (2002), 5975.

[4] David Doty, *Randomized self-assembly for exact shapes*, Tech. Report 0901.1849, Computing Research Repository, 2009.

[5] Martin Fowler, *Language workbenches: The killer-app for domain specific languages?*, June 2005, `http://martinfowler.com/articles/languageWorkbench.html`.

[6] Ming-Yang Kao and Robert T. Schweller, *Reducing tile complexity for self-assembly through temperature programming*, Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006), Miami, Florida, Jan. 2006, pp. 571-580, 2007.

[7] Eric Klavins, *Directed self-assembly using graph grammars*, In Foundations of Nanoscience: Self Assembled Architectures and Devices, Snowbird, UT, 2004.

[8] James I. Lathrop, Jack H. Lutz, Matthew J. Patitz, and Scott M. Summers, *Computability and complexity in self-assembly*, Proceedings of The Fourth Conference on Computability in Europe (Athens, Greece, June 15-20, 2008), 2008.

[9] James I. Lathrop, Jack H. Lutz, and Scott M. Summers, *Strict self-assembly of discrete Sierpinski triangles*, Theoretical Computer Science **410** (2009), 384–405.

[10] Matthew J. Patitz and Scott M. Summers, *Self-assembly of decidable sets*, Proceedings of The Seventh International Conference on Unconventional Computation (Vienna, Austria, August 25-28, 2008), 2008.

[11] ———, *Self-assembly of discrete self-similar fractals (extended abstract)*, Proceedings of The Fourteenth International Meeting on DNA Computing (Prague, Czech Republic, June 2-6, 2008). To appear., 2008.

[12] Paul W. K. Rothemund, *Theory and experiments in algorithmic self-assembly*, Ph.D. thesis, University of Southern California, December 2001.

[13] Paul W. K. Rothemund and Erik Winfree, *The program-size complexity of self-assembled squares (extended abstract)*, STOC '00: Proceedings of the thirty-second annual ACM Symposium on Theory of Computing (New York, NY, USA), ACM, 2000, pp. 459–468.

[14] David Soloveichik and Erik Winfree, *Complexity of compact proofreading for self-assembled patterns*, The eleventh International Meeting on DNA Computing, 2005.

[15] ———, *Complexity of self-assembled shapes*, SIAM Journal on Computing **36** (2007), no. 6, 1544–1569.

[16] Erik Winfree, *Simulations of computing by self-assembly*, Tech. Report CaltechCSTR:1998.22, California Institute of Technology.

[17] ———, *Algorithmic self-assembly of DNA*, Ph.D. thesis, California Institute of Technology, June 1998.

[18] Erik Winfree and Renat Bekbolatov, *Proofreading tile sets: Error correction for algorithmic self-assembly.*, DNA (Junghuei Chen and John H. Reif, eds.), Lecture Notes in Computer Science, vol. 2943, Springer, 2003, pp. 126–144.